Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC

Francisco D. Igual
Texas Advanced Computing Center
figual@tacc.utexas.edu

Murtaza Ali,
Arnon Friedmann
Eric Stotzer
Texas Instruments
{mali,arnon,estotzer}@ti.com

Timothy Wentz
University of Illinois
at Urbana Champaign
wentz1@illinois.edu

Robert A. van de Geijn
Department of Computer Science
The University of Texas at Austin
rvdg@cs.utexas.edu

***Abstract*—Take a multicore Digital Signal Processor (DSP) chip designed for cellular base stations and radio network controllers, add floating-point capabilities to support 4G networks, and out of thin air a HPC engine is born. The potential for HPC is clear: It promises 128 GFLOPS (single precision) for 10 Watts; It is used in millions of network related devices and hence benefits from economies of scale; It should be simpler to program than a GPU. Simply put, it is fast, green, and cheap. But is it easy to use? In this paper, we show how this potential can be applied to general-purpose high performance computing, more specifically to dense matrix computations, without major changes in existing codes and methodologies, and with excellent performance and power consumption numbers.**

***Index Terms*—Low-power architectures, DSPs, linear algebra.**

I. Introduction

A decade ago, it was recognized that Graphics Processing Units (GPUs) could be employed for general purpose high-performance computing. They were relatively easy to program, highly efficient, and, importantly, economy of scale made them affordable. Now, GPUs are an integral part of the scientific computing landscape.

Digital Signal Processors (DSPs) have traditionally been at the heart of embedded systems. Of importance to the HPC community is that they are very low power, benefit from economy of scale, and they are easy to program, provided one is willing to forgo Fortran. Unlike traditional GPUs, they are not accelerators that require a host processor and they hence do not necessarily suffer from the overhead associated with transfering data between the host and the accelerator. They are fully functional cores that are quite similar, from the point of view of programming and memory hierarchy, to a conventional processor. The problem was that, until recently, such processors utilized fixed-point rather than floating-point computation. This has now changed.

We report on our experience with a specific DSP, the Texas Instruments (TI) Keystone multi-core Digital Signal Processor (DSP) architecture, codenamed C66x. The specific multicore processor based on this architecture, namely C6678, studied in this paper, represents the industry's first combined floating-point performance of 128 single precision (SP) GFLOPs (billions of floating point operations per second) on a single device running at 1 Ghz, a power consumption of 10W per chip, and thus with a theoretical ratio of 12.8 GFLOPS/Watt (SP). Its ubiquitous use (they are present in a wide variety of embedded systems, network devices, and similar applications) makes it affordable and available. Its programming model (C/C++ with OpenMP support for multi-threaded codes) will make the port of existing codes straightforward. The fact that it is a standalone processor makes it attractive for applications for which hardware accelerators are not an option. The question is: does it live up to these promises?

Dense linear algebra libraries are often among the first libraries to be ported to a new architecture. There are at least two reasons for this: (1) Many applications cast computation in terms of dense linear algebra options and (2) If an architecture cannot support dense linear algebra libraries effectively, it is likely not going to support other HPC applications well. The `libflame` library [18] is a modern alternative to the widely used LAPACK library. We examine how well it can be mapped to the C66x.

A prerequisite for high performance for dense matrix library is the high-performance implementation of matrix-matrix multiplication kernels known as the level-3 BLAS [5]. The GEneral Matrix-Matrix multiplication (GEMM) operation is simultaneously the most important of these and highly representative of how well the rest of the level-3 BLAS can be mapped to a processor [19], [7], [8]. Since no BLAS library for the C66x existed when we started this study, we describe in detail the implementation of GEMM on this architecture. We will show that, as of this writing, it achieves roughly 10 GFLOPS/core and 74 GFLOPS aggregate on 8 cores, which transforms multi-core DSPs into a highly energy-efficient HPC architecture.

Parallelizing operations to multiple cores requires a thread library and/or compiler support. When this project commensed, there was no such support, but OpenMP support was being added to the TI's C/C++ compiler. Our implementation of a multi-threaded version of GEMM and the `libflame` library provides us with an opportunity to also report on early experiences with the OpenMP implementation by TI.

II. The C6678 DSP architecture

The C6678 DSP is a high-performance fixed/floating-point DSP based on TI's KeyStone multicore architecture [17]. This device incorporates eight C66x DSP cores and runs at a core speed of 1 GHz with a power dissipation of 10W.

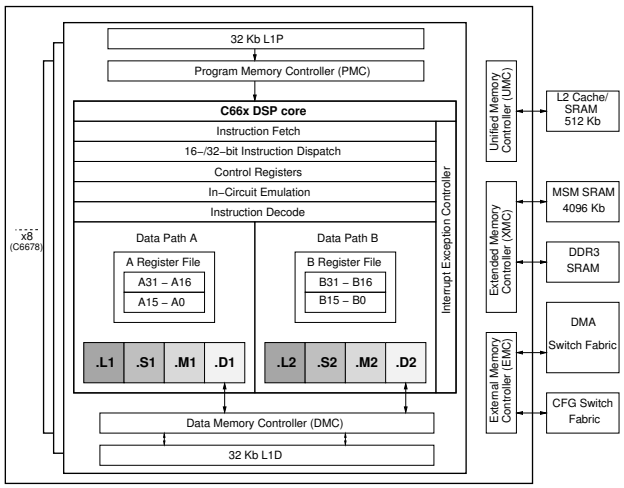

Fig. 1. C66x DSP Core Block Diagram.

A. C66x core capabilities and instruction set

The C66x DSP core is based on Very Long Instruction Word (VLIW) architecture [16]. Figure 1 shows the functional diagram of an individual C66x core. The core has 8 functional units arranged in two sides. Each unit is capable of executing one instruction per cycle. The four unique units on each side are termed as L, M, S, and D units. The M units primarily perform mulitplication operations. The D units performs load/store and address calculation. The additions and various logical and branch operations are distributed between the L and S units. The result is an 8-way VLIW machine where up to eight instructions can be issued in parallel in a single cycle. There are two general-purpose register files (A and B) each containing 32 32-bit registers for a total of 64 registers. Each register file is connected to one side though there is a cross-connect between each register file on one side and the set of units on the other side. The art of programming this architecture involves feeding instructions to as many of the units as possible without overwhelming the registers and the cross-connects. In the case of SGEMM, as will be shown later, the kernel function keeps all the functional units busy every cycle.

The instruction set also includes SIMD instructions which allow vector processing operating on up to 128-bit vectors. The M unit, in particular, can do 4 single precision mulitplies in each cycle. Each of the L and S units can perform 2 single precision additions in each cycle. With two sets of L, S and M units, the core is thus capable of performing 8 single precison multiply-add operations per cycle. The capability of double precision operations are approximately one quarter to that of the single precision operations in the current generation of the architecture. The programmer can also perform various mixed precision operations. All floating point operations are IEEE754 compliant.

B. Memory

The C6678 DSP integrates 32KB of L1 program (L1P) and 32KB of L1 data (L1D) cache as well as 512 KB of L2 cache per core. Both the L1 data and L2 memory can be configured as RAM or cache or part RAM/part cache. The device also integrates 4096KB of Multicore Shared Memory (MSMC in the rest of the paper) accessible by all cores. All L2 memories incorporate error detection and error correction. For fast access to external memory, this device includes a 64-bit DDR3 external memory interface running at 1600 MHz and has ECC DRAM support. The flexibility in choosing cache/RAM for the on-chip dedicated per core memory is fully exploited for intelligent data movement in the core SGEMM implementation.

C. Programmability and multi-thread support

TI's DSPs run a lightweight real time native operating system called SYS/BIOS. Because SYS/BIOS can be used in such a wide variety of different microprocessors with very different processing and memory constraints, it was designed to be highly configurable.

TI provides a C/C++ compiler as part of its Code Generation Tools. In practice, virtually every C89-compliant C program can be directly ported to run on the DSP with no additional effort. To improve the efficiency of the generated code for each TI architecture, the compiler provides a rich variety of optimization and tuning flags. It supports optimization techniques in form of `pragmas` and also intrinsics to extract all the potential performance of the underlying architecture. OpenMP 3.0 is the preferred programming model for C66x multicore DSPs as of today. This allows rapid ports of existing multi-threaded codes to the multi-core DSP, as we illustrate in this paper. TI's C66x compiler translates OpenMP into multithreaded code with calls to a custom runtime library. The runtime library provides support for thread management, scheduling, and synchronization. TI has implemented a runtime library that executes on top of SYS/BIOS and interprocessor communication (IPC) protocols running on each core of the DSP. The TI OpenMP runtime library performs the appropriate cache control operations to maintain the consistency of the shared memory when required. Because TI's C66x multicore DSPs have both local private and shared memory they map well into the OpenMP framework. Shared variables are stored in shared on-chip memory while private variables are stored in local on-chip L1 or L2 memory. Special precaution must be taken to keep data coherency for shared variables, as no hardware support for cache coherency between cores is provided. Some of these issues will be illustrated in Section IV.

III. GEMM ON A SINGLE C66X CORE

Developing an optimized BLAS (Basic Linear Algebra Subprograms [6]) library is usually a first step toward the development of higher level scientific libraries when a new HPC architecture emerges. It also illustrates the performance that can be effectively extracted from the new architecture.

In addition, these kernels are commonly useful to illustrate typical optimization techniques particular to the architecture.

The lack of a Fortran compiler in the TI toolchain makes it even difficult to test a basic reference BLAS implementation. Our trivial first approach was to take the Fortran reference implementations of the BLAS from Netlib, translating these into C using `f2c`, and to utilize the TI C compiler to get the full BLAS running on one core of the DSP.

As is well-known, these reference implementations do not attain high performance, even more on specific-purpose architectures like the TI DSP. To overcome this, an optimized version of single precision matrix-matrix multiplication (SGEMM) has been implemented and is described in this section. Note that GEMM is a key routine towards the development of a full Level-3 BLAS library [11]. Our algorithm is structured much like the implementation in the GotoBLAS [8], [7] library. GotoBLAS is a widely used and high performing BLAS implementation. Since its early versions, it has been successfully ported to many microarchitectures. We illustrate next how Goto's approach can be effectively implemented on the TI DSP.

A. The GotoBLAS approach

Memory hierarchy is commonly divided into multiple levels, each one with a particular size/performance ratio (smaller, faster memories are found as one gets closer to the processor, following a "pyramidal" approach.) The TI DSP is no exception. Following this assumption, GotoBLAS aims at decomposing GEMM into different layers, each one mapped to a different level of the memory hierarchy, with the final goal of amortizing data movements between memory levels with computation and thus attaining near-optimal performance by keeping recently used data as close to the processor as possible.

A typical high-performance GEMM implementation follows the three nested loop approach shown in Figure 2. Considering the operation $C := AB + C$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ matrices, we partition the operands:

$$A = \left(\begin{array}{c|c|c|c} A_0 & A_1 & \cdots & A_{K-1} \end{array} \right), \quad B = \left(\begin{array}{c} \frac{B_0}{B_1} \\ \hline \vdots \\ \hline B_{K-1} \end{array} \right),$$

where A_p and B_p contain b_k columns and rows, respectively (except for A_{K-1} and B_{K-1} which may have less columns or rows, respectively), as shown in the outer loop of Figure 2. We consider this partition to proceed through dimension K. With this partitioning scheme,

$$C := A_0 B_0 + A_1 B_1 + \cdots + A_{K-1} B_{K-1} + C.$$

GotoBLAS implements GEMM as a sequence of highly optimized updates $C := A_p B_p + C$, usually referred as *panel-panel multiplications*, or GEPP attending to the shape of the operands. The performance of the GEPP operation will ultimately determine the performance of the overall GEMM implementation.

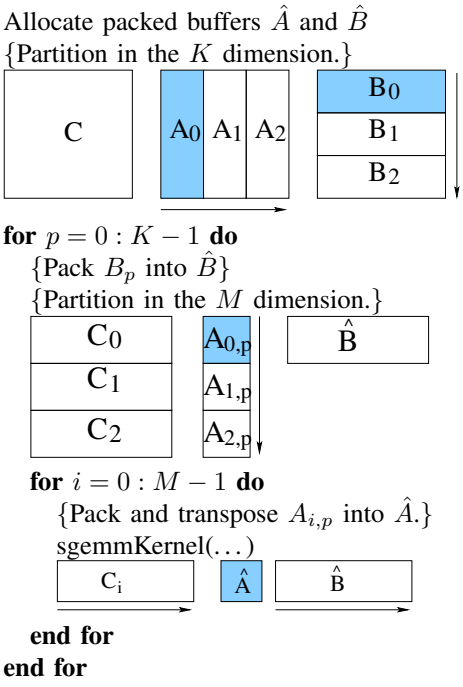

Fig. 2. Basic algorithm for GEMM.

The GEPP algorithm proceeds by packing the corresponding panel B_p in the outer loop into a contiguous memory buffer, partitioning the panel A_p into roughly square blocks $A_{0,p}, A_{1,p}, \ldots, A_{M-1,p}$ and packing them in a similar fashion, as indicated in Figure 2. This proceeds through dimension M. The computation of the multiplication of a block $A_{i,p}$ and the panel B_p is cast in terms of an inner *block-panel multiplication* kernel (GEBP) using the previously packed buffers.

In summary, each GEPP operation requires three different basic building blocks, in which the developer must invest much of the optimization effort:

- An inner kernel (GEBP) that computes $C_i := A_{i,p} B_p + C_i$, where both the block $A_{i,p}$ and the panel B_p are stored in packed buffers.
- Packing and transposing each block $A_{i,p}$ into a contiguous memory buffer $\hat{A}$. $A_{i,p}$ is usually a sub-matrix part of a bigger matrix, and thus it is not contiguously allocated in memory. The packing into a contiguous buffer reduces the number of TLB entries to be accessed. The transposition usually favors the memory access pattern of the inner kernel.
- Packing each panel B_p into a contiguous memory buffer $\hat{B}$, as it is reused for many GEBP calls.

Typically, the dimensions of the block $A_{i,p} \in \mathbb{R}^{m_c \times k_c}$ are chosen so that $\hat{A}$ occupies as much of the cache memory as possible. This often means half the L2 cache, to ensure that accesses to data from matrices B and C do not evict data from this block of A. Intuitively, the closer $\hat{A}$ is to the processor, the faster data movements are. This idea suggests that $A_{i,p}$ should be placed in the L1 cache during the corresponding update. However, Goto demonstrated that loading data from the L2

cache is sufficiently fast that this block can be placed in that layer instead. Since then $m_c \times n_c$ can be larger, the movement of data between RAM and caches is amortized better over actual computation [8]. The general principle is now to place block $\hat{A}$ in the largest cache level available, provided that storing $\hat{A}$ in this cache level allows the computation of $C_i := \hat{A}\hat{B} + C_i$ be computed at the peak rate of the processor. For most architectures, with some care, elements of $\hat{A}$ can be prefetched so that the inner kernel can compute at near-peak performance, when $\hat{A}$ resides in the L2 cache. Thus, the chosen cache level for storing $\hat{A}$ is typically the L2 cache.

The inner kernel GEBP proceeds by partitioning the panel $\hat{B}$ into smaller sub-panels $\hat{B}_0, \hat{B}_1, \ldots \hat{B}_{N-1}$. We consider this partition to proceed through dimension N. The size of $\hat{B}_j$ makes it possible for it to be stored in the smaller L1 cache as it is being multiplied by $\hat{A}$.

Thus, at the deepest level of computation, the overal operation is reduced to a successive set of multiplications of a block, $\hat{A}$, residing in L2 cache by a sub-panel of $\hat{B}$, $\hat{B}_j$, streamed through the L1 cache.

B. Mapping GotoBLAS to the C66x core

Our tuned implementation of a GEMM kernel for the TI DSP follows the same high-level guidelines reviewed in Section III-A. However, the architecture differs from conventional CPUs and their memory in that it allows greater control over the placement of different data sections in physical memory. Determinism necessary for real-time applications for which DSPs were initially designed motivated the addition of these mechanisms to the architecture. As a result, each level of the cache hierarchy can be configured as a fully user-managed scratchpad memory that becomes key to extending Goto's ideas to the DSP.

1) Managing the memory hierarchy: Goto's approach writes the GEBP so as to trick the architecture into keeping $\hat{A}$ and panels of B and C in specific memory layers while they are being used for computations. With the DSP architecture and TI toolchain, we can *force* the linker to allocate given memory sections in a certain physical level of the memory hierarchy (L1 cache, L2 cache, MSMC or DDR3 RAM). This assures that a given buffer will reside on L1 cache, L2 cache or MSMC SRAM during the execution of the kernel.

The usual workflow when developing a code for the TI DSP proceeds in three main steps:

1) *Definition of the memory architecture.* Defines the architectural properties of the target platform, including memory sections, physical allocation of those sections and other parameters such as section size and address range. In our case, we define three different sections mapped to L1 and L2 caches and MSMC SRAM, with names `L1DSRAM`, `L2SRAM`, and `MSMCSRAM`, respectively, with the desired sizes.
2) *Linker configuration.* Provides information to the linker to bind data sections in the source code to memory sections defined in the above memory architecture definition. In our configuration file, we can define three different regions for allocation in L1 and L2 caches, and MSMC SRAM:

```
/* Create .myL1 section mapped on L1 cache */
Program.sectMap[ ".myL1" ] = "L1DSRAM";
/* Create .myL2 section mapped on L2 cache */
Program.sectMap[ ".myL2" ] = "L2SRAM";
/* Create .myMSMC section mapped on MSMC */
Program.sectMap[ ".myMSMC" ] = "MSMCSRAM";
```

3) *Use of scratchpad buffers in the code.* In our GEMM implementation, we guide the compiler to allocate memory for packed buffers into the desired memory section, by using `pragma` directives in the code. With the previous definitions, it is possible to allocate static arrays (`pL1`, `pL2` and `pMSMC`) in different levels of the hierarchy:

```
/* L1 allocation */
#pragma DATA_SECTION( pL1, ".myL1" );
float pL1[ L1_SIZE ];

/* L2 allocation */
#pragma DATA_SECTION( pL2, ".myL2" );
float pL2[ L2_SIZE ];

/* MSMC allocation */
#pragma DATA_SECTION( pMSMC, ".myMSMC" );
float pMSMC[ MSMC_SIZE ];
```

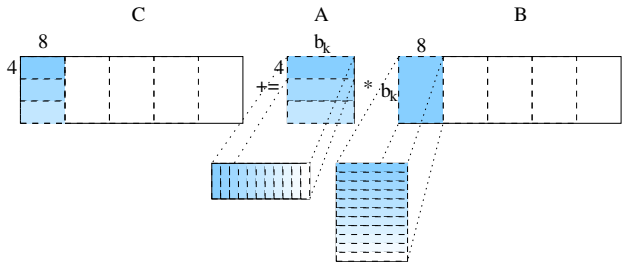

Fig. 3. Inner kernel GEBP implementation proceeds by partitioning $\hat{A}$ and $\hat{B}$ into panels of rows and columns, respectively (top). Each $4 \times b_k$ by $b_k \times 8$ update is performed in the L1 cache as a sequence of highly tuned 4×1 by 1×8 rank-1 updates (bottom).

With these considerations, the implementation of GEMM for the DSP follows the same sequence of outer panel-panel and inner block-panel multiplications described for the Goto approach, using each buffer to allocate the corresponding sub-blocks of A and B.

Most of the effort when developing a new GEMM implementation using this methodology resides in two basic building blocks: the inner kernel and the packing routines. In fact, many of the particularities of the architecture can be hidden in these two routines, maintaining a common higher-level infrastructure. We focus next on the particularities of the TI DSP architecture for the implementation of the inner kernel and packing routines.

2) Optimizing the inner GEBP kernel: The implementation of the GEBP kernel is key to the final performance of the GEPP implementation, and it is usually there where specific optimizations are applied for each architecture. Our implementation partitions both the block, $\hat{A}$, and the panel, $\hat{B}$, into smaller sub-blocks of rows and panels, respectively, as shown in Figure 3.

The computation of a new panel of columns of C is divided

into a sequence of multiplications involving a few rows of $\hat{A}$ (stored in L2 cache) and the current block of columns from panel $\hat{B}$ ($\hat{B}_j$, stored in L1 cache). This preliminary work computes with four rows of $A_{i,p}$ (four columns of $\hat{A}$) at a time, multiplying a block $\hat{B}_j$ with 8 columns. This appears to balance the number of required registers and available parallelism. Before the computation, we prefetch the small sub-block of $\hat{A}$ into the L1 cache, so the deepest level of the overall computation is effectively performed using exclusively data that reside in L1.

The design of the inner kernel implements techniques that exploit many of the particular features of the C66x core to attain high efficiency. Figure 5 shows an actual code extract from the kernel used in our implementation; the kernel proceeds by iterating in the k dimension and performing a rank-1 update at each iteration to update a part of the result matrix C.

The maximum GFLOPS rate offered by the C66x core is attained by the use of vector datatypes and instructions in the inner kernel. The code in Figure 5 illustrates some of the vector datatypes available applied to the calculation of each rank-1 update. `__x128_t` is a container type for storing 128-bits of data and its use is necessary when performing certain SIMD operations. When the compiler puts a `__x128_t` object in the register file, the `__x128_t` object takes four registers (a register *quad*). Similarly, the `__float2_t` container type is used to store two floats. This object are filled and manipulated using specific intrinsics.

The SIMD capabilities of the architecture can be exploited in our code by two different vector intrinsics operating on floating-point data, `CMPYSP` and `DAADDSP` to perform floating-point multipication and addition, respectively:

- *Multiplication*: The C66x core can perform eight single-precision multiplications per cycle: `CMPYSP` instruction can calculate four pairs of single-precision multiplies per .M unit per cycle. Perform the multiply operations for a complex multiply of two complex numbers a and b. Both sources are in 64-bit format. The result is in 128-bit format and contains the following results:

$$c3 = a[1] * b[1]; c2 = a[1] * b[0];$$
$$c1 = -a[0] * b[0]; c0 = a[0] * b[1];$$

  The `CMPYSP` instruction was actually designed for an intermediate step in complex multiplication which gives it the negative sign for one of the multiplications.
- *Addition/substraction*: The C66x core can perform eight single-precision addition/subtraction per cycle: `DADDSP` and `DSUBSP` can add/sub 2 floats and they can be executed on both .L and .S units.

Each rank-1 update proceeds by loading the corresponding elements of $\hat{A}$ and $\hat{B}$ into registers, and accumulating the results of each intermediate SIMD multiplication in the corresponding accumulators that will then be used to update the corresponding entries of matrix C (the update of C is not shown in the code sample), following the schema in Figure 4. The usage of a register pair to reference elements of $\hat{A}$ and the `volatile` definition of these pointers forces the compiler to

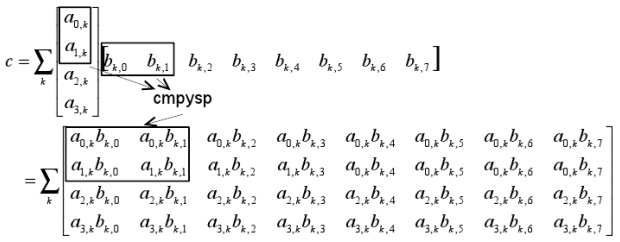

Fig. 4. Basic rank-1 update in the computation of GEBP using SIMD instructions (`cmpysp`).

use load instructions for both registers, and thus make full use of the `.D` units. In summary, each iteration executes 8 `lddw` (double load), 8 `cmpysp` (2x2 outer product), and 8 `daddsp` (double add), and uses 32 registers to store the corresponding entries of C. Four cycles per loop iteration are required, for a total of 16 FLOPS/cycle, and a full occupation of units during the kernel execution.

3) Overlapping communication and computation. Memory requirements, DMA and packing: The DMA support in the TI DSP allows us to hide the overhead introduced by data movement between memory spaces by overlapping communication and computation. Our approach proceeds as described in Figure 6(a), which illustrates a GEPP (the multiplication of a panel of columns of A_p by a panel of rows of B_p) that initially reside in DDR memory and are stored in column-major order. As previously described, this panel-panel multiplication proceeds by dividing A_p into blocks $A_{0,p}, A_{1,p}, \ldots, A_{M-1,p}$, iteratively performing block-panel multiplications of $A_{i,p}$ by the panel B_p.

In order to overlap computation and communication between memory layers, we use double-buffering and we take benefit of the whole memory hierarchy as illustrated in Figure 6(a). We create scratchpad buffers in the three cache levels, that will host sub-blocks of A and B:

- *L1 cache.* A buffer will host a *packed* subblock of the panel $\hat{B}$ with dimensions $b_k \times 8$, necessary as part of the block-panel multiplication. Successive sub-blocks of the panel $\hat{B}$ are streamed to L1 to complete the block-panel multiplication.
- *L2 cache.* A buffer will allocate the complete *packed* block $\hat{A}$ with dimensions $b_m \times b_k$, during the complete computation of the block-panel multiplication.
- *MSMC SRAM.* We create three different buffers that will be used to receive *unpacked* data from DDR RAM through DMA transfers. The first one, with dimensions $b_m \times b_k$, will receive the *unpacked* block of $A_{i+1,p}$ from DDR. The other two, with dimensions $b_n \times b_k$, will host sub-panels of the panel $\hat{B}$, and and will allow the overlap of communication and computation in the block-panel multiply.

The final goal of our algorithm is to multiply $\hat{A}$, residing in L2 cache memory, by a small packed block of $\hat{B}$ that resides in L1. In the first step of a panel-panel multiply, the algorithm starts two DMA transfers of the block $A_{0,p}$ and the first sub-

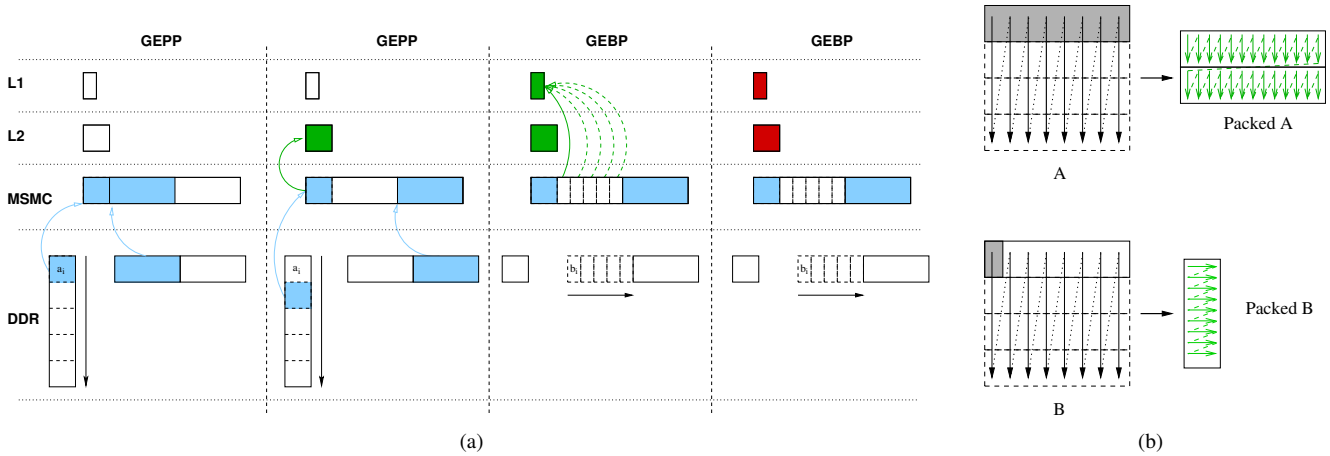

Fig. 6. (a) Illustration of the overlapping of DMA transfers to on-chip memories and actual computation in an iteration of a GEPP operation. Blocks in blue indicate pending DMA transfer. Blocks in green indicate packed buffers. Blocks in red indicate actual computation. (b) Packing (and transposing if necessary) of sub-matrices of A and B.

panel of B to the corresponding locations in MSMC SRAM. Upon the completion of the transfer, $A_{0,p}$ is packed and stored in the corresponding L2 buffer, and the corresponding buffer in MSMC SRAM becomes empty and ready to receive $A_{1,p}$. At this point, the communication of $A_{1,p}$, necessary for the next block-panel multiplication and the next columns of $\hat{B}$, necessary for the current block-panel multiplication, to MSMC SRAM are started. These two transfers and the multiplication occur simultaneously. The multiplication of $\hat{A}$ by a sub-panel of B, with $\hat{A}$ packed in L2 cache and the sub-panel of B unpacked in MSMC SRAM, proceeds by sequentially streaming small sub-blocks of B to the buffer in L1 cache, packing the sub-block in the format required by the inner kernel. Once both a block of A and the small sub-block of B (both packed) reside in L2 and L1 cache, respectively, the inner kernel proceeds as described in the previous section. The packing pattern of blocks of A and B is illustrated in Figure 6(b) and performed by the CPU (not using the DMA engine).

IV. Multi-core DSP programming

Developing a multi-threaded BLAS implementation is one approach to build efficient multi-threaded higher level dense linear algebra libraries. We propose two different approaches to extract parallelism and build efficient multi-threaded libraries for dense linear algebra on the DSP. The first one aims at manually developing multi-threaded codes for each particular BLAS operation. The second one is based on a runtime system, SuperMatrix, that automatically extracts task parallelism and only requires a sequential BLAS implementation to build multi-threaded codes.

A. *Explicit multi-threaded* GEMM

In this first multi-threaded implementation, each thread updates a panel of rows of matrix C. Thus, each thread multiplies a panel of rows of A by the whole matrix B. Each local computation follows the same approach as that illustrated for one core. From the point of view of memory, we replicate the buffer structure allocated in MSMC memory, as each thread packs its own block of A_p and panel of B_p independently. L1 and L2 buffers are local to each core and thus do not have to be replicated. DMA mechanisms are still in place, using different DMA resources for each core, and thus overlapping communication and calculation at core level. As each thread operates on different blocks of C, no cache coherency concerns apply, besides a final flush (*write back*) of data from local caches to DDR RAM.

A more scalable approach [12], which also avoids some duplicate packing of data in B, is beyond the scope of this initial study.

B. *Runtime-based BLAS*

Task parallelism has demonstrated its suitability for extracting parallelism for multi-threaded dense linear algebra libraries [15], [1] or other types of codes [14] and architectures. We combine several related concepts to automatically extract parallelism and effectively parallelize sequential codes. First, we store matrices *by blocks*, instead of using a column-wise or row-wise storage scheme; we consider a block as the basic unit of data. Second, we derive and implement *algorithms-by-blocks*. An algorithm-by-blocks executes a sequential algorithm on blocks, being a single operation with a block the basic unit of computation. The algorithm evolves by performing smaller sub-problems on sub-matrices (blocks) of the original matrix. Third, we consider the *data dependencies* between tasks to automatically keep track of ready tasks and map them to available execution resources, much in the line of a superscalar processor.

The `libflame` library integrates a mechanism, called SuperMatrix, that leverages algorithms-by-blocks to automatically parallelize sequential codes and schedule sub-operations (or *tasks*) to a multi-threaded architecture. This runtime system operates by extracting and maintaining a directed acyclic graph (DAG) of tasks, that includes information about tasks that

```
void sgemmKernel(const float *pA, const float *pB,
                 float *pC, const float a, const int k,
                 const int stepC)
{
  __float2_t s0, s1, s2, s3, s4, s5, s6, s7;
  __float2_t s8, s9, sa, sb, sc, sd, se, sf;
  __float2_t * restrict ptrB     = (__float2_t *) pB;
  // Twin addresses defined volatile so the compiler
  // use load instructions for both registers;
  // If not, it optimizes out one address thereby reducing
  // the usage of D units at the expense of an LS unit;
  volatile __float2_t * restrict ptrA
                        = (__float2_t *) pA;
  volatile __float2_t * restrict ptrATwin
                        = (__float2_t *) pA;

  s0 = 0.0; s1 = 0.0; ...  se = 0.0; sf = 0.0;

  for (index = 0; index < k; index++)
  { // Loop over k; Each iteration performs rank-1 update
    __float2_t b01, b23, b45, b67, a01, a23,
               a01Twin, a23Twin;
    __x128_t   reg128;

    a01 = *ptrA++; a23 = *ptrA++;
    // compiler is using LS units to create
    // a twin register
    // D units are available;
    // force a load (that is, a D unit)
    // to use twin register;
    a01Twin = *ptrATwin++; a23Twin = *ptrATwin++;

    b01 = *ptrB++; b23 = *ptrB++;
    b45 = *ptrB++; b67 = *ptrB++;

    reg128 = _cmpysp(b01, a01);
    // Accumulate b[0]*a[1] and -b[0]*a[0]
    s0   = _daddsp(s0, _lof2_128(reg128));
    // Accumulate b[1]*a[0] and b[1]*a[1]
    s1   = _daddsp(s1, _hif2_128(reg128));

    reg128 = _cmpysp(b23, a01);
    // Accumulate b[2]*a[1] and -b[2]*a[0]
    s2   = _daddsp(s2, _lof2_128(reg128));
    // Accumulate b[3]*a[0] and b[3]*a[1]
    s3   = _daddsp(s3, _hif2_128(reg128));

    reg128 = _cmpysp(b45, a01Twin);
    // Accumulate b[4]*a[1] and -b[4]*a[0]
    s4   = _daddsp(s4, _lof2_128(reg128));
    // Accumulate b[5]*a[0] and b[5]*a[1]
    s5   = _daddsp(s5, _hif2_128(reg128));

    reg128 = _cmpysp(b67, a01Twin);
    // Accumulate b[6]*a[1] and -b[6]*a[0]
    s6   = _daddsp(s6, _lof2_128(reg128));
    // Accumulate b[7]*a[0] and b[7]*a[1]
    s7   = _daddsp(s7, _hif2_128(reg128));

    // Proceed similarly with a23 and a23Twin
    // to get s0...sf
    ...
  }
  // Copy accumulators to output matrix
  ...
}
```

Fig. 5. Code for the inner kernel used in GEBP.

compose a dense linear algebra operation and data dependencies between them. At runtime, only when a task sees all its input data dependencies satisfied, can be scheduled for execution to a free computational unit. After the execution of a task, the coresponding dependency information is updated in the DAG if necessary, releasing new tasks that become ready for execution. The existence of a set of worker threads waiting for ready tasks makes it possible to execute many data-independent tasks in parallel, and thus to exploit the inherent task parallelism in a given dense linear algebra algorithm. What is more important, this parallelization is carried out without any changes in the existing sequential algorithm-by-blocks. Besides performance, the main advantage of this approach is *programmability*, as existing, thoroughly tested sequential codes are automatically mapped to multi-threaded architectures.

We note that PLASMA [1] provides a similar runtime system which, more recently, now includes the kind of out-of-order scheduling that SuperMatrix has supported since its inception.

C. Previous SuperMatrix implementations

The SuperMatrix runtime was originally designed and developed for SMP architectures, using OpenMP or `pthreads` as the underlying thread support [3]. In the original design, a pool of threads maintains a common list of ready tasks in a producer-consumer fashion, executing tasks as they become ready. Work-stealing, data-affinity and cache-affinity techniques have been developed to increase both load balancing and data locality, with excellent performance and scalability [2].

Flexibility is at the heart of the `libflame` library in general, and the SuperMatrix runtime in particular. The full funcionality of the library has been successfully ported to a wide variety of commercial or experimental architectures. They include the Intel SCC experimental platform, with 48 x86 cores [10], or platforms with multiple Nvidia GPUs [15], [9]. In this accelerator-based architecture, we deploy as many *host threads* (running on the host, or CPU) as accelerators exist in the system, and consider each accelerator as a unique computation unit. Thus, each time a task eligible for execution on the accelerator is found, necessary input data is transferred to the corresponding memory space (usually through the PCIe bus), the task is executed on the corresponding accelerator using an optimized parallel library (in the case of Nvidia, invoking CUBLAS kernels), and data is retreived back to host memory as necessary, where the hybrid execution continues. `libflame` was the first dense linear algebra library to port all its functionality to systems with multiple GPUs.

D. Retargeting SuperMatrix to multi-core DSPs

Although an accelerator-based implementation as that developed for multi-GPU architectures will be possible for future systems with multiple DSPs attached to a host system through the PCIe bus, we advocate here for an alternative implementation in which the whole runtime is running inside the DSP cores, without a host. This way, the DSP architecture and runtime system acts as a fully standalone linear-algebra embedded system, exploiting all the benefits of this type of hardware from the point of view of the ubiquity and power efficiency.

The main problem when porting the SuperMatrix runtime to the multi-core DSP is the lack of hardware support for cache coherency between caches from different cores. While cache coherency has been managed by software in other SuperMatrix

implementations [4], the total absence of cache coherency in the TI DSP adds a new burden considering the whole runtime logic is run in the DSP, without a cache-coherent host attached to it. As a result, in the OpenMP implementation provided by TI, the user is in charge of managing coherence of shared data.

We can define two different types of shared data in the internals of `libflame`: *control data* (lists of pending and ready tasks, dependency information, etc.) and *matrix data* (the actual matrix data). The dynamic nature of the runtime system makes it necessary to heavily use dynamic memory allocation for these types of data structures. While *matrix data* coherency can be managed by software in a relatively easy way, the complexity of handling *control data* coherency becomes a real burden and translates into a dramatic increase in the complexity of SuperMatrix internals.

The TI software stack provides an elegant and flexible workaround for this problem: as we show with statically allocated arrays, it is also possible to define different *heaps* to store dynamically allocated memory, and to map these heaps to different memory sections. In addition, mechanisms for enabling or disabling caching on given memory pages allow us to create heaps in cached or non-cached memory. Following the workflow illustrated for a single core:

1) *Definition of the architecture.* In this case, we create two different memory sections in our architecture, both mapped on physical DDR3 memory, with names `DDR3_NOCACHE` and `DDR3`, to allocate the uncached and cached heaps, respectively.
2) *Linker configuration.* Consider the following extract of configuration file:

```
/* Create a Default Heap on MSMCSRAM. */
BIOS.heapSize                    = 0x40000;
BIOS.heapSection                 = "systemHeap";
Program.sectMap["systemHeap"]    = "MSMCSRAM";

/* Create a Non-cached Heap on DDR3. */
var UnCached                     =
                          new HeapMem.Params();
UnCached.size                    = 0x1000000;
UnCached.sectionName             = "UnCached_heap";
Program.global.UnCached_heap     =
                          HeapMem.create(UnCached);
Program.sectMap["UnCached_heap"] = "DDR3_NOCACHE";

/* Create a Cached Heap on DDR3. */
var Cached                       =
                          new HeapMem.Params();
Cached.size                      = 0x10000000;
Cached.sectionName               = "Cached_heap";
Program.global.Cached_heap       =
                          HeapMem.create(Cached);
Program.sectMap["Cached_heap"]   = "DDR3";

/* Set DDR uncached section */
Cache.setMarMeta(0x80000000, 0x8000000, 0);
```

   With this configuration, we define three different heaps, each one with its own properties: in the default heap, with a size of 256 Kb, will be allocated that memory reserved with classic `malloc` calls. The second and third ones are special heaps, and must be managed through special routines provided by the TI software stack that allow the management of dynamic memory on specific heaps. We define two different ad-hoc heaps with 16 Mb and 256 Mb, respectively, and map them to the memory sections named as `DDR3_NOCACHE` and `DDR_CACHE`. If we consider that the section named as `DDR3_NOCACHE` starts at adress `0x8000000`, the instruction `Cache.setMarMeta` disables the cacheability of the memory pages comprised in its address range. Proceeding this way, we have defined a heap mapped on DDR3 memory whose memory transactions bypass the cache subsystem, and thus cache coherency does not become a problem.
3) *Code.* With this configuration, the modifications in the runtime code are limited to the memory management module in `libflame`. We provide routines for allocating/deallocating memory from different heaps depending on the type of data we are managing (non-cached *control data*, or cached *matrix data*).

To avoid the performance penalty due to the use of non-cached DDR memory, in practice we map the non-cached heap to MSMC on-chip memory instead of DDR3. Thus, the SuperMatrix runtime allocates *control data* in non-cached fast on-chip memory (MSMC), and *matrix data* in cached slow memory (DDR).

Proceeding this way, the runtime system only deals with cache coherency issues related to *matrix data*. Consider the usual runtime operation. Whenever a new ready task is extracted from the DAG, the corresponding kernel is assigned to a worker thread and executed in the corresponding core (e.g. in the case of a GEMM task, the optimized kernel described in Section III is executed). Upon the execution of the task, the runtime system must *invalidate* the copies of input matrix blocks that would remain in cache (blocks of A and B in the case of GEMM), and *write-back and invalidate* the copies of output matrices that have just been written (blocks of C in the case of GEMM). Proceeding this way, future tasks working on the same blocks on the same or a different core will always encounter consistent versions of the corresponding operands.

With only these considerations, not only GEMM but the full functionality of `libflame` has been ported to the multi-core TI DSP without major code changes. This includes algorithms-by-blocks and different algorithmic variants for Level-3 BLAS and LAPACK functionality (Cholesky, LU factorization with incremental pivoting, QR factorization, eigenvalue solvers, ...) [9]. What is more important, the usage of a DSP is transparent to both the library developer of new linear algebra routines and the user codes that make use of them. The full performance benefits of this full functionality will not be reaped until more single-core BLAS functionality is available.

This design permits the utilization of the C6678 DSP as a purely standalone system, without the need of an external host orchestrating the parallel execution. The benefits are clear: lower acquisition cost, lower power consumption, the possibility of running HPC applications on fully embedded architectures, no data transfer penalties, and code reutilization from existing SMP software solutions. However, if an accelerator-based system (like a multi-GPU architecture) is desired, the SuperMatrix runtime supports that model ully by implementing the same software architecture as was designed

for multi-accelerator systems [9].

V. Experimental results

A. Hardware setup

All the experimental evaluation was carried out using a TMDXEVM6678LE evaluation module (EVM). This EVM includes an on-board C6678 processor running at 1 GHz, equipped with 512 MB of DDR3 RAM memory. To improve the programming productivity and reduce the development time, we employed a XDS560V2 emulation board, which greatly reduces the time for loading code. On the software side, Code Composer Studio version 5.1 was used. This includes SYS/BIOS version 6.32 as the RTOS operating system, Code Generation Tools version 7.3.0 (C/C++ compiler), MCSDK version 2.1, and TI's prototype implementation of the OpenMP runtime in an internal pre-release version. We used single precision in all our experiments. Peak for double precision computation on the current architecture is roughly 1/4 that of single precision computation. It is expected that a future version of this architecture will deliver peak double precision performance that is half that of single precision performance.

B. GEMM performance on one core

Figure 7 reports the performance attained by our optimized GEMM implementation on one C66x core. Results are reported for square matrices in terms of GFLOPS. The y-axis is adjusted so that the top of the graph represents the theoretical peak performance of one C66x core (16 GFLOPS). We show the performance for $C = AB$ (NN), $C = AB^T$ (NT), $C = A^T B^T$ (TT), and $C = A^T B$ (TN). The maximum performance attained is 9.84 GFLOPs, for the case in which both A and B are not transposed. The slightly lower performance for the other cases is mainly due to different packing procedures depending on the exact layout of the original matrices in DDR3 memory. These numbers suggest an actual utilization of the core of around a 61%, much in line of modern, tuned BLAS implementations for GPUs, although still far from the 95+% attained by general-purpose CPUs. Asymptotic performance is attained for relatively small matrices (around $n = 512$).

Four sources of inefficiency exists in our inner kernel code, many of them directly related to architectural issues:

- L1 cache bank conflicts occur at every iteration in our main kernel. An increase in the number of L1 cache banks would reduce this issue.
- Loop overhead and load of C to memory: we have observed some corner cases where the cache for matrix C evicts the cache for the stack. Looping over larger panel would help, but this is directly limited by the amount of L1 memory.
- Cache misses occur at the loading stage of each panel of A into cache memory which happens for every GEBP kernel call.
- Additional cache misses also occur related to data movement between memory layers. The number of misses is directly dependent on the specific SGEMM case and data size, and it is directly related to the size of L2 and MSMC memory.

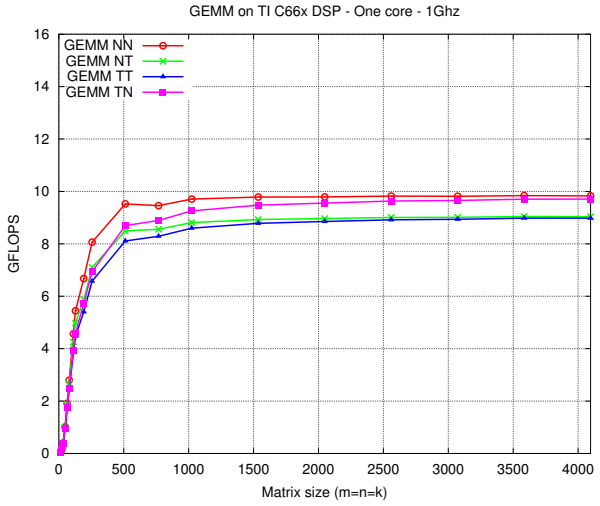

Fig. 7. Matrix-matrix multiplication performance on one C66x core for square matrices.

Further refinement of the described techniques may overcome some or all of these inefficiencies.

C. GEMM performance on multiple cores

Figure 8 reports the multi-core performance results attained by our two alternative multi-threaded implementations of GEMM. On the left, we show the performance attained by a manually parallelized OpenMP code. On the right, we show the performance attained by our SuperMatrix port. Both plots are adjusted to the theoretical peak performance of GEMM on 8 cores based on the single-core implementation. Results are given in terms of GFLOPS for an increasing number of cores (from 1 to 8), for the GEMM case in which both A and B are square matrices and are not transposed. For the SuperMatrix evaluation, we have carried out a thorough evaluation of performance varying the block size used by the algorithm-by-blocks. In the figure, we only report performance for the optimal one.

The usage of MSMC memory to store the runtime *control data* limits the amount of on-chip memory that can be effectively used as part of the inner kernel. While in the manually parallelized implementation all MSMC memory can be devoted to actual computation data, in the runtime-based implementation this amount of memory must be reduced to allow the runtime logic to fit in on-chip memory. This fact limits the performance of the inner kernel and, thus, the overall performance of the parallel implementation. As a reference, in Figure 8 we add the performance of the manually parallelized GEMM on 8 cores using the exact kernel and memory setup as utilized in the runtime-based implementation, labelled as `8c - Manual`.

Peak performance on 8 cores is 74.4 GFLOPS for the manually parallelized GEMM. For this implementation, the attained

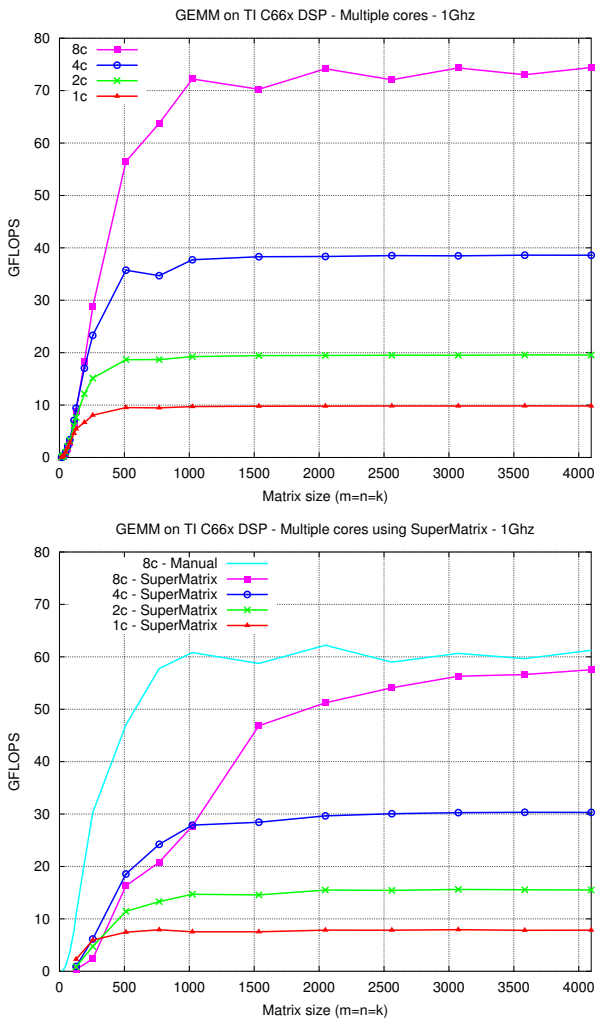

Fig. 8. Matrix-matrix multiplication performance on one to eight C66x cores for square matrices using a manual parallelization (top) and a runtime-based system (bottom).

speedups for the largest tested matrices are 1.99, 3.93, and 7.59 for 2, 4, and 8 cores compared with the optimal sequential implementation, respectively; for the runtime-based GEMM, the attained speedups for the largest tested matrices are 1.92, 3.82, and 7.25. These results demonstrate the scalability of both software solutions towards multi-threaded dense linear algebra implementations on the multi-core DSP. The differences in raw performance between both approaches are justified by the different memory setups used for each implementation. The overhead of using a runtime system orchestrating the automatic parallelization can be observed for relatively small matrices, for which the manual approach is clearly more efficient. For larger matrix sizes, the runtime system is fairly competitive and achieves similar peak performance.

D. Power efficiency

Table 9 reports a summary of the power efficiency of common current HPC architectures, both of general purpose

Architecture	GFLOPS	$\frac{\text{GFLOPS}}{\text{W}}$	Utilization
Core i7-960	96	1.2	95%
Nvidia GTX280	410	2.6	66%
Cell	200	5.0	88%
Nvidia GTX480	940	5.4	70%
Stratix IV	200	7.0	90%
TI C66x DSP	74	7.4	57%

Fig. 9. Performance of commercial HPC architectures running SGEMM. Source: [13]

(Intel multi-core) or specific purpose (Nvidia GPUs, Cell B.E. or FPGAs) in terms of GFLOPS/Watt. We take the maximum performance for single precision GEMM found in the literature for each architecture as a reference [13]. In the case of the DSP, we report numbers for our best parallel implementation, and consider the peak performance of the architecture as 128 GFLOPS to obtain the utilization ratio.

With these results, the TI DSP exhibits the best GFLOPS/Watt ratio for our implementation of SGEMM, improving on that attained by the last generation of Nvidia GPUs, Intel multi-cores and even FPGAs. Moreover, many of the architectures listed in the table are merely hardware accelerators, hence they need additional hardware support to work as a functional system. Our DSP-based implementation is a purely standalone system, with no need of an external host and therefore with no extra power requirements.

VI. Conclusions

We have introduced the new TI C66x multi-core DSP as a novel architecture that fits the demands of today's HPC. We have discussed its strengths (common programming models for sequential and parallel codes, control over memory allocations, flexibility of the architecture, powerful compiler, ...) and weaknesses (lack of cache coherency between cores) of the architecture. We have demonstrated how it can be efficiently and easily programmed by adopting successful ideas from general-purpose CPUs in the dense linear algebra arena, and illustrated some common optimization techniques with actual pieces of code.

We have described an efficient implementation of GEMM on one DSP core and two different strategies for exploiting the multiple cores in the DSP. On one core, the GotoBLAS-like approach illustrated for GEMM has been recently applied to the rest of the BLAS-3 routines to obtain the first complete high-performance BLAS-3 implementation on a DSP. On multiple cores, the runtime-based option allows an easy adaptation of existing algorithms-by-blocks to this novel parallel architecture. As far as we know, this is the first runtime system aiming at automatically parallelize sequential codes fully ported to a DSP. Performance results are impressive, as is scalability and power efficiency. This makes the TI DSP one of the most efficient current solutions, competitive even with the excellent GFLOPS/Watt ratio of modern Nvidia Fermi GPUs.

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, Vol. 180, 2009.

[2] E. Chan. *Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations*. PhD thesis, Department of Computer Sciences, The University of Texas, August 2010.

[3] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007. ACM.

[4] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP'08*, 2008.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[7] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[8] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3), 2008.

[9] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. Van Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, (0):–, 2011.

[10] F. D. Igual, G. Quintana-Ort, and R. van de Geijn. Scheduling algorithms-by-blocks on small clusters. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2012.

[11] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model, implementations and performance evaluation benchmark. LAPACK Working Note #107 CS-95-315, Univ. of Tennessee, Nov. 1995.

[12] B. A. Marker, F. G. Van Zee, K. Goto, G. Quintana-Ortí, and R. A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing*, pages 748–757, February 2007.

[13] A. Pedram, R. van de Geijn, and A. Gerstlauer. Co-design tradeoffs for high-performance, low-power linear algebra architectures. Technical Report UT-CERC-12-02, UT Austin, CERC, 2011.

[14] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[15] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, 2009.

[16] TMS320C66x DSP CPU and Instruction Set Reference Guide. `http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf`, November 2010. Texas Instruments Literature Number: SPRUGH7.

[17] TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor. `http://www.ti.com/lit/ds/sprs691c/sprs691c.pdf`, February 2012. Texas Instruments Literature Number: SPRS691C.

[18] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. The libflame library for dense matrix computations. *Computing in Science and Engineering*, 11:56–63, 2009.

[19] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11. IEEE Press, 2008.